Mandatory exercise for INF144

Øyvind Hytten

March 19, 2015

Java compiling and running

Compiling:

```
javac -encoding "UTF-8" Mandatory1.java
```

Running:

```
java Mandatory1
```

Follow the instructions on screen. All functionality is implemented into subclasses within this class.

The program is specifically written for Windows as it reads from and writes to file directories on a Windows format.

All file operations performed by this program happens within the directory 'G:\inf144\ohy092\mandatory1\' by default, which can be changed either through a user query after running, or in a static field variable before compiling.

Markov modeling

First off, I've had some difficulties actually understanding how to evaluate the Markov statistics. I think I got this in order, but when it came to generating the random text samples I could not quite figure out how to properly handle one special case; the obvious, initial case in which we have no recorded suffix for the initial prefix, because the initial prefix (that is, the start of our generated text) is null (or empty). This case can also be extended to comprise all states in which our generated text has not yet reached the length of our suffixes.

After some reasoning, I realized something that perhaps should be obvious also: our initial string needs to be a valid prefix. I therefore decided to also record "starting prefixes", i.e. prefixes that start with a capital letter. I collect these in an ArrayList, and these entries are not necessarily unique. Finally, I pick one recorded starting prefix uniformly at random (uniformly in the sense that for instance "Jeg" and "Jeg" are two different entries if they have been recorded twice, so there is some weighting on our starting prefix selection).

I am not sure if this is the correct way to go about this problem, but it is what I decided seem like a natural solution. If I had somehow picked a starting prefix that is not a valid prefix in the sense that we have no previous recording of it, then we would get a NullPointerException while looking for the next character in our generated sequence.

I also handle order zero approximation as a special case, as this is actually just picking characters at random with the same probability distribution as they appear in the input text. For all other orders of approximation my code functions in the same way, and by simply changing the code to accept larger values for this order as user input it would work for any order smaller than the input text size.

As I mentioned earlier, for each new text generated I read the input text and generate its Markov chain for the given order of approximation. Let us call this order o. Then, for all substrings of length o - that is, all possible prefixes - I record the preceding character and append it to the list of possible suffixes for the given prefix. The Markov model is represented as a HashMap containing a number of prefixes, each with an ArrayList containing their possible suffixes. All prefixes and suffixes are added in their lower case. This is done within the method createModel.

This list of possible suffixes is not unique, meaning that the prefix "ask" can have the letter "e" recorded as a possible suffix several times. This provides a weighting when we later generate texts based on this model. If the prefix starts with an upper case letter I also record this as a starting prefix, which I can later use to start building a randomly generated texts. These starting prefixes are also not necessarily unique.

For the special case of order 0 I record all (lower case) characters as possible suffixes for an empty string prefix.

For generating a new, random text, I retrieve one of the recorded starting prefixes as our base. I then append this prefix with a letter previously recorded as a possible suffix for the given prefix, and repeat this process using the very last substring of length o as our prefix in each iteration, until we reached our desired length which is the same as the length of our input text. This is done in the generateText and nextChar methods.

Finally, the output is written to a file.

Questions from the task specification

1. Do you recognize some real words in the random text?

For an order 0 of approximation, what comes out is plain gibberish. With an order 1, a few words make sense, but the output is mostly unreadable.

For larger orders of approximation, yes. Several words are actual words. When I test with even larger orders of approximation than what the task specifies (i.e. order 6), not only are more or less all of the words real words - except for words that are already misspelled in the source file - but parts of the text actually provide some semantical meaning. It is not good literature, but adjacent words tend to "fit" each other with these larger orders of approximation.

2. Are the Markovian information sources unifilar, and if so explain why they are unifilar and compute their entropy.

For an order 0 of approximation, they are not. An order 0 approx. is in this case equivalent to computing the statistical probabilites for all possible characters, and generating a new text "tossing in" these characters based on their computed distribution. This means there are no states from which to compute our next character, or that there is only one state regardless of the text we have generated so far.

For orders other than 0, the information sources are unifilar due to the fact that each state (single, double or triple character depending on the order) has its own transition probability distribution.

The entropy of each source is

$$\sum_{i=1}^{n} w_i \cdot h(p_i) = \sum_{i=1}^{n} w_i \cdot ((-p_i)\log_2(p_i) - (1-p_i)\log_2(p_i))$$

with a Markov model containing n possible transition states for the particular order of approximation.

However, I've tried computing the asymptotic probabilities w for order 1 by computing its transition matrix, but my results did not make sense. Hence i am not able to compute the source entropy for neither this order, nor the larger ones.

LZW (de-)compression

Compression
I start by building a dictionary mapping some character sequence to a new char value. Ideally, the first new recorded char sequences should be represented only by b bits, where b is the number of bits needed to represent the initial dictionary plus one new entry (in this case b would be $\log_2(30 + 1)$), but I found this hard to implement, and went with this less-optimal solution, which unfortunately does not work for an infinite number of char sequences.

For each new symbol (character sequence not previously recorded - or rather unrecorded pair of previously recorded symbols) I find, I add this as a new entry in the dictionary, and append the first symbol's key (of the pair) to our compressed string. I repeat this until the source string is empty.

I also map all characters to the lowest possible value, i.e. instead of 'a' corresponding to the ASCII value 97, as it usually does, it now corresponds to 0, etcetera.

Decompression
I re-create the dictionary by mapping the first 30 indices to our initial dictionary characters, then adding new entries as I discover pairs from the source and using these entries to build up the string as I find them traversing the source string.

I did have some problems regarding reading and writing to file this way, but I solved it using an BufferedInputStream instead of a BufferedReader.

Huffman (de-)compression

This solution is not optimal either. I pondered this for a while. I found no comprehensible way to rebuild the generated Huffman dictionary without any prior knowledge about it, so I decided to prepend it to the compressed text. This will of course increase the space needed to represent it, but it also raises another issue: which kind of delimiter can we use in order to distinguish the dictionary from the encoded data? And for that matter, we also need a delimiter distinguishing character (key) values from binary strings (as ones and zeroes can also be key values).

I decided to solve this by prepending an integer followed by a single space in both cases. This way, we know how many of the preceding characters are to be considered as either a string representation of the dictionary (in the first case) or as a binary Huffman code (in the second case). This, however, also increases the space we need to represent our text somewhat.

Interestingly, however, the program reports different compression ratios for Huffman if I compile and run it through cmd or Eclipse. In the first case, ratio seems to improve over LZW, whereas in the latter it is negative, meaning it actually consumes more space than plain text. The negative ratio is due to my prepended dictionary representation, but I do not know why this differs subjective to the method of compilation and running.

Compression

I build the Huffman tree by creating the nodes for each symbol read (as leaves), and creating adding them to what initially is a forest. A long as this forest contains more than one distinct tree, I consequently find the two internal nodes with the lowest frequencies and add them together in a new internal node until the tree is complete.

After this, I generate the code by traversing the tree downwards, appending a 0 as I go left and a 1 as I go right. When I hit a leaf I add the sequence (Huffman code) to the Huffman dictionary.

When the dictionary is complete, I replace all symbols with their corresponding Huffman codes to a bit string. Since bytes are represented by bits in groups of 8, I prepend some zeroes and a one to the bit string in order to make its length divisible by 8. This way, I know on the "receiving side" that our actual bit string starts after the first one. I parse this bit string to a "regular" string, which is our encoded (compressed) message, and prepend the string representation of the dictionary.

Decompression

This is fairly easy once the dictionary is prepended. I re-build the dictionary by reading the specified number of characters as a new string, reading the first character of this string and the preceding integer (with a following space), then reading the character's corresponding bitwise Huffman code. I remove the characters I just read from the string, and repeat until the string is empty.

Last, I parse our encoded string as a bit string, and find all entries matching any given Huffman code, and add its value to the decoded string.

Compression ratios

When compiling and running in Eclipse, my program reports the following compression ratios, which I believe to be correct:

```
Before compression:
Total size: 3196000, Compression ratio 0,0%
Order 0 size: 799000, Compression ratio 0,0%
Order 1 size: 799000, Compression ratio 0,0%
Order 2 size: 799000, Compression ratio 0,0%
Order 3 size: 799000, Compression ratio 0,0%
```

```
After LZW compression:
Total size: 1450844, Compression ratio 54,6%
Order 0 size: 419480, Compression ratio 47,5%
Order 1 size: 359013, Compression ratio 55,1%
Order 2 size: 336673, Compression ratio 57,9%
Order 3 size: 335678, Compression ratio 58,0%
```

```
After LZW and Huffman compression:
Total size: 4474597, Compression ratio -40,0%
Order 0 size: 1287222, Compression ratio -61,1%
Order 1 size: 1106136, Compression ratio -38,4%
Order 2 size: 1066289, Compression ratio -33,5%
Order 3 size: 1014950, Compression ratio -27,0%
```

Ratios are relative to plain text size in all cases, and tell us how big a percentage of the plain text source we managed to strip away.

The "order o size" column tells us the total size (in number of characters generated in the Java code) for the full 100 texts of order o of approximation. "Total size" is simply the sum of these values for all orders.

As expected, ratios decrease with larger orders of approximation due to their increased likelihood of repeating sequences/symbols.